(GIGA)n SCIENCE

COMMENTARY Open Access

Ten recommendations for software engineering in research

Janna Hastings*, Kenneth Haug and Christoph Steinbeck

Abstract

Research in the context of data-driven science requires a backbone of well-written software, but scientific researchers are typically not trained at length in software engineering, the principles for creating better software products. To address this gap, in particular for young researchers new to programming, we give ten recommendations to ensure the usability, sustainability and practicality of research software.

Keywords: Software engineering, Best practices

Background

Scientific research increasingly harnesses computing as a platform [1], and the size, complexity, diversity and relatively high availability of research datasets in a variety of formats is a strong driver to deliver well-designed, efficient and maintainable software and tools. As the frontier of science evolves, new tools constantly need to be written; however scientists, in particular early-career researchers, might not have received training in software engineering [2], thus their code is in jeopardy of being difficult and costly to maintain and re-use.

To address this gap, we have compiled ten brief software engineering recommendations.

Recommendations

1. Keep it simple

Every software project starts somewhere. A rule of thumb is to *start as simply as you possibly can*. Significantly more problems are created by over-engineering than under-engineering. Simplicity starts with design: a clean and elegant data model is a kind of simplicity that leads naturally to efficient algorithms.

Do the simplest thing that could possibly work, and then double-check it really does work.

*Correspondence: hastings@ebi.ac.uk
Cheminformatics and Metabolism, European Molecular Biology Laboratory – European Bioinformatics Institute, Wellcome Trust Genome Campus, CB10 1SD Hinxton, UK

2. Test, test, test

For objectivity, large software development efforts assign different people to test software than those who develop it. This is a luxury not available in most research labs, but there are robust testing strategies available to even the smallest project.

Unit tests are software tests which are executed automatically on a regular basis. In test *driven* development, the tests are written first, serving as a specification and checking every aspect of the intended functionality as it is developed [3]. One must make sure that unit tests exhaustively simulate *all possible* – not only that which seems reasonable – inputs to each method.

3. Do not repeat yourself

Do not be tempted to use the copy-paste-modify coding technique when you encounter similar requirements. Even though this seems to be the simplest approach, it will not remain simple, because important lines of code will end up duplicated. When making changes, you will have to do them twice, taking twice as long, and you may forget an obscure place to which you copied that code, leaving a bug.

Automated tools, such as Simian [4], can help to detect and fix duplication in existing codebases. To fix duplications or bugs, consider writing a library with methods that can be called when needed.

4. Use a modular design

Modules act as building blocks that can be glued together to achieve overall system functionality. They hide the

```
public static int doSomethingToAnInt(int n) {
    boolean doIt = (((n & (n-1)) == 0)?false:true);
    while ((n & (n-1)) != 0) n = (n & (n-1));
    return (doIt?n<<1:n);
}
```

Figure 1 An example of incomprehensible code: What does this code actually do? It contains a bug; is it easy to spot?

details of their implementation behind a public interface, which provides all the methods that should be used. Users should code – and test – to the interface rather than the implementation [5]. Thus, concrete implementation details can change without impacting downstream users of the module. Application programming interfaces (APIs) can be shared between different implementation providers.

Scrutinise modules and libraries that already exist for the functionality you need. Do not rewrite what you can profitably re-use – and do not be put off if the best candidate third-party library contains more functionality than you need (now).

5. Involve your users

Users know what they need software to do. Let them try the software as early as possible, and make it easy for them to give feedback, via a mailing list or an issue tracker. In an open source software development paradigm, your users can become co-developers. In closed-source and commercial paradigms, you can offer early-access beta releases to a trusted group.

Many sophisticated methods have been developed for user experience analysis. For example, you could hold an interactive workshop [6].

6. Resist gold plating

Sometimes, users ask for too much, leading to feature creep or "gold plating". Learn to tell the difference between essential features and the long list of wishes users may have. Prioritise aggressively with as broad a collection of stakeholders as possible, perhaps using "gamestorming" techniques [7].

Gold plating is a challenge in all phases of development, not only in the early stages of requirements analysis. In its most mischievous disguise, just a little something is added in every iterative project meeting. Those little somethings add up.

7. Document everything

Comprehensive documentation helps other developers who may take over your code, and will also help you in the future. Use code comments for in-line documentation, especially for any technically challenging blocks, and public interface methods. However, there is no need for comments that mirror the exact detail of code line-by-line.

It is better to have two or three lines of code that are easy to understand than to have one incomprehensible line, for example see Figure 1.

Write clean code [8] that *you* would *want* to maintain long-term (Figure 2). Meaningful, readable variable and method names are a form of documentation.

Write an easily accessible module guide for each module, explaining the higher level view: what is the purpose of this module? How does it fit together with other modules? How does one get started using it?

8. Avoid spaghetti

Since GOTO-like commands fell justifiably out of favour several decades ago [9], you might believe that spaghetti code is a thing of the past. However, a similar phenomenon may be observed in inter-method and inter-module relationships (see Figures 3 and 4). Debugging – stepping through your code as it executes line by line – can help you diagnose modern-day spaghetti code. Beware of module designs where for every unit of functionality you have to step through several different modules to discover where the error is, and along the way you have long lost the record of what the original method was actually doing or what the erroneous input was. The use of effective and granular logging is another way to trace and diagnose problems with the flow through code modules.

9. Optimise last

Beware of optimising too early. Although research applications are often performance-critical, until you truly encounter the wide range of inputs that your software

```
public static int roundUpToNearestPowerOfTwo(int n) {
    //Operation only supported for positive integers
    if (n<0) throw new IllegalArgumentException("roundUpToNearestPowerOfTwo: n must be positive ("+n+")");

    // if n is a power of 2, then the bitwise sum of the bits in n and n-1 will be 0
    // e.g. 00001000 & 00000111 (no 1-bits in common positions)
    if ((n & (n-1)) == 0) return n;

    // n is not a power of two, so find the most significant power of two below n
    while ((n & (n-1)) != 0)
        n = (n & (n-1));

    //multiplying that by 2 gives the nearest higher power of 2 above the original n
    return n*2;
}
```

Figure 2 This code peforms the same function, but is written more clearly.

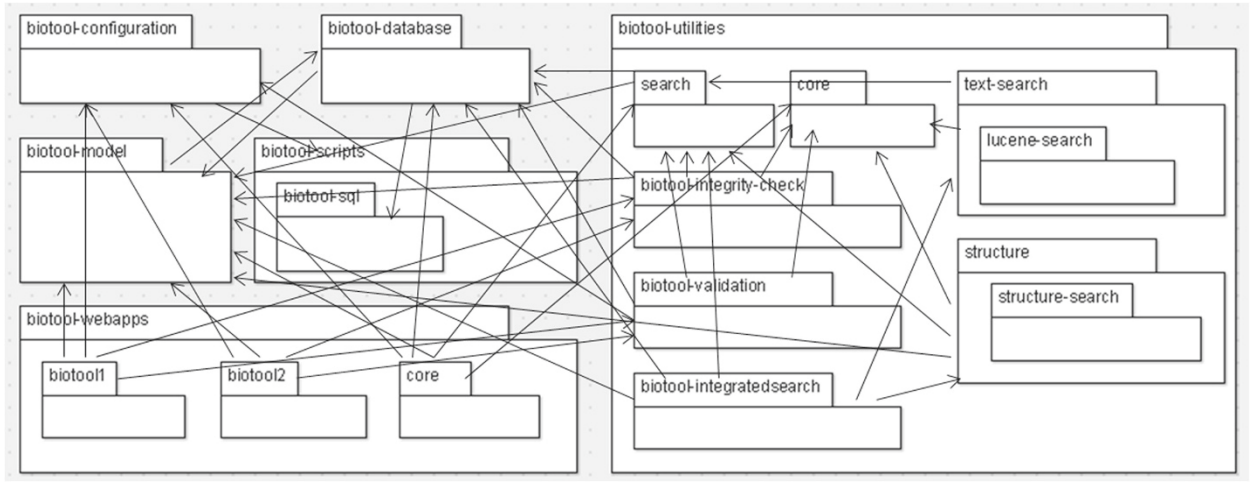

Figure 3 An unhealthy module design for 'biotool' with multiple interdependencies between different packages. An addition of functionality to the system (such as supporting a new field) requires updating the software in many different places. Refactoring into a simpler architecture would improve maintainability.

will eventually run against in the production environment, it may not be possible to anticipate where the real bottlenecks will lie. Develop the correct functionality first, deploy it and then continuously improve it using repeated evaluation of the system running time as a guide (while your unit tests keep checking that the system is doing what it should).

10. Evolution, not revolution

Maintenance becomes harder as a system gets older. Take time on a regular basis to revisit the codebase, and consider whether it can be renovated and improved [10]. However, the urge to rewrite an entire system from the beginning should be avoided, unless it is really the only option or the system is very small. Be pragmatic [11] – you may never finish the rewrite [12]. This is especially true for systems that were written without following the preceding recommendations.

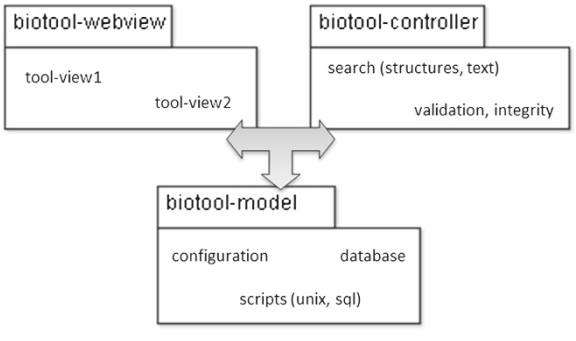

Figure 4 The functional units from the biotool architecture can be grouped together in a refactoring process, putting similar functions together. The result may resemble a Model-View-Controller architecture.

Use a good version control system (e.g., Git [13]) and a central repository (e.g., GitHub [14]). In general, commit early and commit often, and not only when refactoring.

Conclusion

Effective software engineering is a challenge in any enterprise, but may be even more so in the research context. Among other reasons, the research context can encourage a rapid turnover of staff, with the result that knowledge about legacy systems is lost. There can be a shortage of software engineering-specific training, and the "publish or perish" culture may incentivise taking shortcuts.

Table 1 Further reading

Description	URL
Software Carpentry: scientific computing skills; learn online or in face-to-face workshops	http://software-carpentry.org/
The Software Sustainability Institute	http://software.ac.uk/
Learn more about what makes code easy to maintain	http://www.thc.org/root/phun/unmaintain.html
How to write good unit tests	http://developer.salesforce.com/page/How_to_Write_Good_Unit_Tests
What is clean code?	http://java.dzone.com/articles/what-clean-code-%E2%80%93-quotes
Introduction to refactoring	http://sourcemaking.com/refactoring/
The danger of premature optimization	http://c2.com/cgi/wiki?PrematureOptimization

This table lists additional online resources where the interested reader can learn more about software engineering best practices in the research context.

The recommendations above give a brief introduction to established best practices in software engineering that may serve as a useful reference. Some of these recommendations may be debated in some contexts, but nevertheless are important to understand and master. To learn more, Table 1 lists some additional online and educational resources.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
JH prepared the initial draft. All authors contributed to, and have read and approved, the final version.

Acknowledgements
This commentary is based on a presentation given by JH at a workshop on Software Engineering held at the 2014 annual Metabolomics conference in Tsuruoka, Japan. The authors would like to thank Saravanan Dayalan for organising the workshop and giving JH the opportunity to present. We would furthermore like to thank Robert P. Davey and Chris Mungall for their careful and helpful reviews of an earlier version of this manuscript.

Received: 25 September 2014 Accepted: 19 November 2014
Published: 4 December 2014

References

1. Goble C: **Better software, better research.** *IEEE Internet Comput* 2014, **18:**4–8.
2. Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, Guy RT, Haddock SHD, Huff KD, Mitchell IM, Plumbley MD, Waugh B, White EP, Wilson P: **Best practices for scientific computing.** *PLoS Biol* 2014, **12:**e1001745.
3. Beck K: *Test Driven Development*. New York: Addison-Wesley; 2002.
4. Simian: **Similarity Analyzer.** [http://www.harukizaemon.com/simian/]
5. Bloch J: *Effective Java*. New York: Addison-Wesley; 2008.
6. Pavelin K, Pundir S, Cham JA: **Ten simple rules for running interactive workshops.** *PLoS Comput Biol* 2014, **10:**e1003485.
7. Gray D, Brown S, Macanufo J: *Game-storming: A playbook for innovators rulebreakers and changemakers*. Sebastopol, CA: O'Reilly; 2010.
8. Martin RC: *Clean Code: A Handbook of Agile Software Craftsmanship*. New York: Prentice Hall; 2008.
9. Dijkstra EW: **Letters to the editor: go to statement considered harmful.** *Commun ACM* 1968, **11**(3):147–148.
10. Fowler M: *Refactoring: Improving the Design of Existing Code*. Reading, Massachusetts: Addison Wesley; 1999.
11. Hunt A, Thomas D: *The Pragmatic Programmer*. Reading, Massachusetts: Addison Wesley Longman; 2000.
12. Brooks FP: *The Mythical Man Month*. New York: Addison-Wesley; 1975.
13. **The Git distributed version control management system.** [http://git-scm.com/]
14. GitHub: **An online collaboration platform for Git version-controlled projects.** [http://github.com/]

doi:10.1186/2047-217X-3-31
Cite this article as: Hastings *et al.*: Ten recommendations for software engineering in research. *GigaScience* 2014 **3**:31.